5-1992

# The Systems Control Language (SyCoL): A New Language for Distributed Industrial Control

Doug E. Martin

# The Systems Control Language (SyCoL):

# A New Language for Distributed Industrial Control

University Honors Thesis

by Douglas Martin

Bachelor of Arts in Computer Science

Southern Illinois University

Spring 1992

Table of Contents

# Chapter One

## Introduction

In this chapter, a overview of programmable logic controllers in undertaken. It includes their history, advantages, components, and programming languages. The chapter concludes with a statement of reason this research was undertaken.

## History of Programmable Logic Controllers

Until the late 1960's electro-mechanical devices were the main components in industrial control operations. These devices, known as relays, were linked together by the thousands to control sequential manufacturing processes and stand-alone machines. While these relays were reliable in singular form, when they were linked together by hundreds of wires the reliability and maintenance factors became very challenging (Johnson 1).

Along with these considerations came the issue of their high installation cost. Typical configurations, including the parts, wiring, and installation labor, could range from $30 to $50 per relay. To make matters worse, when the control needs of the process changed, it called for a complete rewiring of the relay circuits. This rewiring often took place months later using personnel that were sometimes unfamiliar with the circuit operations and often, if the circuits were poorly documented, the entire relay system was scraped to save time and costs (Johnson 3).

Facing all these problems with relay systems, it was obvious that another technology was needed to replace relays. What was needed was a technology that could withstand the factory environment and be readily changed to fit changing control needs. That technology came in the late 1960's in the form of the programmable logic controller.

## Advantages of Programmable Logic Controllers

The invention of the programmable logic controller (PLC) gave a great boost to high-volume production environments. PLC's provide a system for process engineers that allows for low down-time when control changes needed to be implemented, and also a low down-time when diagnostics and repairs are needed.

The low-down time for control changes is due to the fact that the changes are not made on the physical system level, that is rewiring, but rather they are made at the logical level, in the controllers computer memory. Moreover, this logical rewiring takes place in a fraction of the time need for physical rewiring and also allows the process engineer to quickly fix any errors may have been designed into the system (Johnson 7).

The reason for the low-down time for repair and diagnostics is that the components of the system that could readily physically fail are removed from the control logic. More specifically, the relays that once provided for the circuit logic are replaced with solid state semiconductor logic which has little to no chance of physical failure. This leaves only the components that interface to the process being controlled, and diagnosis of problems with these components is fairly trivial (Johnson 8).

## Components of Programmable Controllers

All PLCs consist of the following four functional blocks: inputs, outputs, central processing unit, and programming device. To understand the operation of the PLC, and thus the control system, each block must be fully explored (Johnson 3).

Inputs to the PLC consist of digital and analog components. Examples include pushbuttons, limit switches, proximity switches, photosensors, theromocouples, position sensing devices, and bar code readers. The signals from these components are converted into meaningful data for the central processing unit.

Outputs of the PLC also consist of digital and analog components. Examples of outputs include pilot lights, display devices, motor starters, DC and AC drives, solenoids, and printers. These components, which are given data by the central processing unit, allow the PLC to control the process and inform the process supervisor of the current state of the controller.

The central processing unit (CPU) is the brain of the PLC. It consists of a microprocessor, logic memory for storing the actual control logic, storage memory for variable data, and a power supply. The specific operation of the microprocessor is beyond the scope of this paper, however a generalized description of its operation will be given.

Basically, the CPU utilizes its logical memory to store the needed information to control a process. After this information is stored, the CPU starts to solve the logic from the start of memory. This process continues until the end of memory is reached, at which time the process starts over at the beginning of memory. This is call "scanning", and it continues in the PLC until the time the power to the PLC is removed.

The final component of the PLC is the programming device. This component, unlike the others, is not used in the operation of the PLC, but as the name suggests, during the program development time. These devices are divided into two classes: dedicated devices, and personal computers. In the beginning dedicated devices were the sole means of programming the PLC. These first consisted of light emitting diode (LED) devices, but were later improved through the use of a cathode ray tubes (CRTs). These dedicated controllers are optimized for usage but suffer from a lack of expandabilty. Recently, manufacturers have been offering an alternative to the dedicated device, which is the personal computer (PC). The PC allows the process engineer to use a combination of software to not only control the process, but to monitor the process and perform quality control operations automatically. Another advantage of using personal computers over dedicated devices is the savings accrued because duplicated dedicated device hardware costs are eliminated (Johnson 4).

## Programming Languages of Programmable Logic Controllers

As mentioned in the last section, there are a number of programming devices available to the process engineer. These devices present an interface between the process engineer and the process to be controlled. The interfaces are usually realized in one of the following four languages: relay ladder logic, function block programming, boolean programming, and special application programming. These programming languages will be discussed in detail in the next chapter, but are outlined here so that I may present the reason for my research into programmable controller languages.

Relay ladder logic is basically an extension of the method that old relay control systems were documented. It consists of a series of graphic symbols representing physical components that are connected together to from a circuit that realizes the control operation needed.

Boolean programming is borrowed from the field of discrete digital design. It consists of symbols representing AND, OR, NOT and other logical operations. These symbols are connected together to realize the control operation.

Special application programming consists of individual languages designed by PLC manufacturers. These languages are usually designed around a type of operation to be performed such as motion control, or continuous production control, but may include general purpose languages (Johnson 20).

## Purpose of This Research

While all of the languages outlined in the previous section performed well in the age of the simple automated factory, they are showing their weakness now as fully integrated factories are coming on-line. In these integrated factories, where planning, production, and distribution are optimized, the use of these languages presents a bottleneck in production speed and efficiency. Current research has focused on taking the human out of the process engineering equation through the use of artificial intelligence. However, industrial researchers are finding out, as computer science researchers have found out, that the flexible modeling of a complex process like control design is extremely difficult, and computationaly expensive.

It is my belief that the human shouldn't be taken out of the process engineering design procedure. I believe that a system that combines modern software engineering techniques with a distributed network architecture would provide a more flexible and responsive control design system. I also believe that such a system would enable management to take a more active role at the plant floor level, both in quality control and quality assurance.

It is with this in mind that I set out to research and design a new PLC language, which this report outlines.

# Chapter Two

## Analysis of Existing Programmable Logic Controller Languages

In this chapter, the PLC languages that were outlined in the previous chapter we be analyzed in-depth so that their strengths and weaknesses can be ascertained. These strengths and weaknesses will be used in the design of my proposed PLC programming language.

## Analysis of Relay Ladder Logic

Today's relay ladder logic is an extension of the old method by which process engineers used to document relay control systems. It uses a series of symbols to represent both physical and logic components, an input line, an output line, and any number of lines connecting the aforementioned components together. The physical components represented by different symbols include motors, lights, pushbuttons, and limit switches. The logical components include addition and subtraction, counters, timers, latches, and subprogram branching. By connecting the components together, the process engineer sequences and controls the process.

Relay ladder logic gives the process engineer a method in which he or she can quickly program a simple control problem. Its use of a graphic symbology allows rapid program construction on personal computers and allows others to quickly understand the program.

The main disadvantage of relay ladder logic is its limited instruction set, as it has no facilities for data logging or statistical analysis. While this may seem to contradict the above statement that a limited graphic set is preferred, it in fact does not. Limiting the graphic set does not have to mean a limiting of the instruction set, as will be seen in my design (Barney 27).

## Analysis of Boolean Logic Programming

Boolean logic programming is borrowed from the field of combinational-sequential digital logic design. It uses AND, OR, and NOT gates of combinational circuitry, and timers, counters, and latches from the sequential side of digital design. It represents these operations using the standard digital design graphical symbols. The logical operations can be shown to be very similar to the relay logic operations, that is, AND is equivalent to two contacts in series, and OR is equivalent to two contacts in series. The timers, counters, and latches are built in the same manner (Barney 45).

This method of programming the PLC allows for flexible specification for the control problem. It lets the process engineer think of the problem in logical terms and thus may give a more bug free solution. Moreover, many products have been developed in the digital design field that the process engineer may use. These products include computer-aided design (CAD) tools, automated circuit generation tools, and automated testing tools.

However, there is a severe drawback associated with thinking in and implementing the control problem in logical terms, and that is program size. A boolean logic program, by definition, uses the most basic components, and thus, it takes a much larger number of these components to specify a

control problem than if the process engineer used relay ladder logic. Now while using these smaller components may reduce program execution time, with today's affordable high-speed computers, this most likely won't matter. Thus, when speed is not a consideration, the use of the larger, more complicated boolean logic methodology needs to be reevaluated.

## Analysis of Special Application Programming

The final language type under analysis is special application programming. This method of programming includes manufacturer specific programming languages such as motion control system languages or data management languages. The method also includes general-purpose programming languages that are modified or supplied with libraries to allow the process engineer to design a solution to the control problem.

These languages let the process engineer look at the control problem as a general computational problem. This view gives the engineer a great deal of flexibility in the implementation of the control solution. The use of a general purpose language also allows the engineer to incorporate data gathering and data analysis sections into the control system. It also allows the engineer to interface multiple machines together into one control system in a more efficient manner than with any other programming interface (Barney 54).

However, there are a number of disadvantages in using a general purpose language for the implementation of a control system. The first of these disadvantages is the fact that the process engineer must learn the syntax and semantics of the programming language. It is a well known fact that the learning curve for a new programming language is a very long one. This slow process of learning the language may be exacerbated further when multiple versions of the language

reside on different machines in the control system. Another disadvantage with general purpose programming languages is that they require the process engineer to program a great deal of the low level functions associated with the control system. This type of programming is well known for its difficulty in writing and debugging. The final significant disadvantage is the fact that there are so many different general purpose languages in existence. This great diversity in languages means that a control system may not be able to be ported to a different machine setup, which cuts significantly into the bottom line.

## Comparisons of the Programming Methods

Now that all three programming methods of the PLC have been looked at, the strengths and weaknesses of the methods can be extracted. This will provide a set of guidelines for the creation and analysis of my PLC programming language.

The primary strength in the relay logic methodology is its ease of use. The language allows the programmer to think in graphical terms, and the use of a limited language set allows other personnel to understand the control solution quickly and easily. The primary strength of the boolean methodology is also its use of a graphical language set, however, the resolution in which the programmer must think of the control problem is too fine, that is, the language constructs are to elementary for effective and efficient control engineering. Finally, when looking at the strengths of a general purpose programming language, one item stands out, flexibility. So with all of these in mind, the chapter ends with the following table that outlines the points that need to be addressed when designing a PLC programming language.

### Design Points

1) Be easy to use
2) Be easy to learn
3) Provide flexibility
4) Provide security
5) Abstract the control solution

# Chapter Three

## The Need of a New Programmable Logic Controller Language

This chapter argues that a new programmable logic controller language needs to be implemented. It bases this argument on the analysis of existing languages contained in the last chapter.

## New Developments in Manufacturing and Computer Science

The last few years have seen significant changes in the field of manufacturing. Computer technology has integrated medium to large scale manufacturing operations in such a manner that the entire manufacturing process is now under computer supervision using a distributed network. This supervision includes inventory control, process control, quality control, and resource planning. In the face of this rapidly expanding technology, the field of programming PLCs has fallen behind -- process plans are being created by artificial intelligence but are being conceptually controlled by connected relays!

The use of an antiquated control programming system is in my view the bottle neck for the creation of a flexible medium to large scale manufacturing system. I also believe that the solution to the problem does not lay in the application of artificial intelligence because control programming is a design problem, and thus it cannot be efficiently computationaly modeled.

Paralleling this growth in the field of manufacturing technology has been the growth of an area of computer science, that is, the field of software engineering. Researchers in this field have be investigating the best methods and interfaces to use in order to produce quality software. The amount of research has been considerable, and

the conclusions drawn have been varied. However, a number of consistent findings have been reported in the field (Sodhi 5).

The researchers have outlined the characteristics of real-time languages, of which PLC programming languages are a subset. The characteristics are security, readability, flexibility, simplicity, portability, and efficiency. These characteristics are the same as the ones extracted in the last chapter from the combination of the existing PLC programming languages. The researchers have also outlined the goals of software engineering, they are reliability, modifiability, maintainability, understandability, adaptability, reuseability, efficiency, portability, tractability. These goals are to be reached through a set of guiding principles. These principles are abstraction, information hiding, completeness, confirmability, modularity, localization, error handling, and uniformity. Finally, these principles are to be incorporated into a programming methodology that insures their preservation. Examples of current methodologies are the structured approach, the object-oriented approach, the entity relation approach, event oriented approach, and the stepwise refinement approach. All these terms will be investigated further in the description of the new language, but are given here to shown the guidelines by which the language was designed (Sodhi 10).

## Combining PLC Technology with Software Engineering

Given the developments outlined in the previous section, I think the best method to advance current PLC technology is to introduce the principles set forth from the field of software engineering. I think that a PLC programming language that is based on the principles of software engineering would give a immediate return in both the efficiency and flexibility of a manufacturing system.

I also believe that if the current method of PLC programming is continued, the newest area of manufacturing research, distributed control, will be slowed significantly. Given this, I have researched and designed a new PLC programming language for a distributed environment that employs all the principles of software engineering but still caters to the needs of the process engineer. The language is entitled SyCoL, for Systems Control Language.

# Chapter Four

## Distributed Environments and the Design of SyCoL

The purpose of this paper is to propose a new language for distributed PLC programming. This language, called SyCoL, for Systems Control Language, was designed using both current research in distributed control theory and computer-aided software engineering techniques. The reason for the combination of the two fields, as well as a general overview of them, will first be discussed. After the basis of the design has been given, an overview and detailed description of SyCoL will be presented along with a example problem.

## Overview of Distributed Control

Before any discussion of a new language for distributed PLC programming, there must be a common agreement as to what exactly is the definition of distributed control, as there are many in current research literature. However, all of the definitions seem to solely differ in the amount and method of communication between the local control units and the host controller. For the purpose of this paper, I am adopting the definition of distributed control as follows --a system of interconnected intelligent programmable controllers which communicate directly to other controllers in the system to aid in efficient system control. Using this definition, all of the advantages of distributed control, as outlined by Lukas, can be realized. The advantages include a reduction in costs for both installation and maintenance, and an increase in amount of modularity, performance, and reliability (Lukas 112).

Given these advantages, which are far greater than the advantages afforded by stand-alone control systems, it is obvious to see that the preferred method of building future industrial systems would be with the use of the distributed paradigm. However, there is a stumbling block on the road to distributed control, and that is the programming languages available to the control engineer. Current efforts in the area of distributed control languages are centered around the

adaptation of stand-alone control languages or the adaptation of general purpose programming languages to the control problem. I believe that the solution to the distributed language issue does not lie in the adaptation of existing languages, but rather in the creation of a new language using a new area in computer science -- computer-aided software engineering.

## Overview of Computer-Aided Software Engineering

Computer-aided software engineering (CASE), as mentioned above, is a new area of study in computer science. It is defined by Lewis as a set of tools that automate the production, maintenance, and distribution of software products (Lewis 1). The method by which these tools operate is to link the "artifacts", as Lewis terms them, which are simply the program listings and documentation of a computer system, to the processes of software engineering, which include the procedures, rules-of-thumb, and interaction among team members (Lewis 1). The advantage of this linkage of process and product is the creation of quality software efficiently and cost effectively.

By creating a CASE tool for distributed systems, I believe that the programming and debugging time of such systems could be drastically reduced. The reason for the reductions in time would be due to the automatic programming of common control situations afforded by the CASE tool, as well as the automatic management of the programs on all of the local control units. This paper is a proposal for the language of just such a tool -- a language called SyCoL.

## Overview of SyCoL

Through the use of CASE tool technology, SyCoL would enable the process engineer to create a control procedure quickly and with a lesser chance of errors. This reduction in design time is be due to SyCoL's use of an intuitive graphical interface. This interface allows the engineer to program the control system by connecting together a series of graphical icons that represent components in the process. It also allows the engineer to add other elements into the control system, such as quality control and quality assurance procedures. Thus, SyCoL not only serves as a control language, but also as a tool for the factory management.

## Description of SyCoL

As mentioned above, SyCoL operates under a graphical environment so that the control program can be implemented in a more intuitive manner. However, the decision to use a graphical environment raises a great deal of questions. These questions include hardware considerations, such as the type of display device to use, software considerations, such as the computer language to use to implement SyCoL, and esthetic considerations, such as how the programming and operator interface should look.

To bypass all these considerations, SyCoL will adopt the XWindows standard for both the programming and operator interface. This standard, which is hardware independent, uses the language C for its programming language, and defines every aspect of its interface with the user. By doing this, the user of SyCoL is insured that once a control system is written, that it may be run on many different computer systems. Also, this allows any third-party vendors to easily design and market extensions to the language, thereby insuring SyCoL's rapid growth and acceptance in the marketplace.

Using this graphical interface, the user begins programming the control system by selecting the inputs and outputs of the system. Inputs could include pushbuttons, strain gauges, position sensors, and outputs could include

physical elements such as motors and lights, or computer elements, such as messages on the computer screen. These components would then be drawn as icons, or pictures that represent what they are, on the screen.

After all the components have been selected, the user begins to connect the components together in the form of a dependency diagram through the use of a mouse device. This diagram is simply a set of directed edges, that is, lines that start at one component and end at either another component or another line. For example, a motor of a sawmill is to turn on when both a safety button is pressed and a position sensor indicates that a log is in position. In this case, the user would first connect the safety button icon to the motor icon. When this is done, a arrow is drawn between the two icons. After this arrow is drawn by the CASE tool, the user would then connect the position sensor icon to the arrow. Thus, the user has now specified that the operation of the motor somehow depends on the operation of both the button and the sensor.

Now that the dependency diagrams have been drawn, the user continues programming the control system by selecting one of the dependency diagrams by selecting it with the mouse. This brings up a new screen that contains the components in the diagram along with arrows from component to component. The user then uses the mouse to select one of the four types of objects, called functional units, that are placed on the arrows between components. The three types of

functional units are routers, which route paths between components, agents, which request information from the component to the left and pass it on to the component to the right, observers, which request information from the component to the left and use it for their own purposes, and actors, which pass information on to the component to the right. By using these four components, any control system can be implemented.

To complete the example system, the user would select the dependency diagram that connects the button, sensor, and motor. Once this is done, a screen is brought up that contains these components along with their arrows. The user would select two agents and place each on the screen. Each agent would be connected on the left side from each component. After connecting, the user would select one of the agents, say for example, the button. By selecting a agent, another screen is brought up in which the user selects a question to ask the button. The question to be asked is selected by the user from a list of pre-defined questions for the component that is connected on the left side. In the case of the button the user would select "Is your button down?", and in the case of the sensor, the user would select "Is there something in front of you?". After setting-up the agent, the user would select a router unit and place it on the screen. Then, the user would connect the arrows from the two agents to the left side of the router, and the arrow from the motor to the right side of the router. The user would

then select the router unit with the mouse. After being selected, another screen would be brought up. In this screen, the user would select the messages that when received, should activate the path to the component on the right side, in this case, the motor. The messages that should activate the motor are a combination of "I am down" and "There is something in front of me." Finally, the user would place an actor between the output of the router and the motor. Then by selecting the actor, the user is able to pick from a list of messages that the motor will accept, in this case the user would pick "Turn on."

Although the last example may seem lengthy, the actual time to implement the system would be just a few minutes, compared to the hours that it might take using any other distributed control programming language. Furthermore, once implemented, the control program can be easily debugged by inserting observers into the control path to see what messages are being passed through the system.

The example just given showed some of the possible uses of the four different functional units. The following table lists some other uses for the function units:

| Unit Type | Operation |
|---|---|
| Router | Logical AND, OR, NOT.<br>IF statement and CASE statement<br>Multiplexed output<br>Encoded input |
| Agent | Exception handler<br>Pre and post condition checking |

| | |
|---|---|
| Observer | Debugging tool<br>Data logging and analysis |

It is important to realize that SyCoL is just a part of the intended distributed control CASE tool. Other elements of the CASE tool include program version control and tracking, component creation, testing and maintenance facilities, and automatic documentation management.

To analyze SyCoL, the requirements set down in the previous chapters concerning software engineering and PLC programming must be reexamined. They specified that the language must:

1) Be easy to use
2) Be easy to learn
3) Provide flexibility
4) Provide security
5) Provide robustness
6) Provide functionality
7) Provide for easy insertion
8) Abstract the control solution

Without going into detail, it can be shown that SyCoL meets all of the above requirements because of the combination of a limited number of language elements with a graphical environment.

# Chapter Five

## Implementing SyCoL

## A New Programmable Logic Controller Language

This chapter analyses the different methods of implementing the SyCoL. I look at both implementation and execution costs.

## The Different Methods of Implementation

Basically, there are three methods of implementation available for SyCoL. The methods are interpretation, translation, and compilation. Each method is the result of a trade-off between program development time and program execution time. The following three sections investigate the pros and cons of each type of implementation in relation to program development and program execution time.

## Interpreting SyCoL

The method in which program development time is minimized and program execution time is forfeited is interpretation. This method gives immediate feedback from the system to the programmer, and thus allows the programmer to debug the system quickly and efficiently. The reason that the execution time is forfeited is described below along with an outline of the interpretation process.

The interpretation process starts with a source file that describes the program in the language to be interpreted. The interpreter then takes this description and reads it into memory in small meaningful amounts. These small amounts are usually single lines in the program. The small amounts of information are individually decoded and checked to see if they are valid statements in the language. If they are indeed valid statements, the corresponding routines that the language statements specify are executed in the computer. After the routines are executed, the process starts over by reading in the next meaningful unit in source file. This process is ended when either the end of the program is reached, an error occurs, or the programmer interupts the process in some pre-defined manner (Aho 34).

In examining the above process, one can see the reason for the slow execution time -- each statement must be individually examined and executed, and many times each

statement may be examined more than once. This lack of "remembering" statements may seem inane, but there is a reason for it. The primary reason is that because the interpreter doesn't need to remember past lines, it is far easier to implement. The second reason is that the programmer, as mentioned above, may arbitrarily stop the program and change it. If the program were to remember lines, it would have to also remember any relationships that they may have to one another as well. This is so it can change any lines that may be affected by the modification, which would be very difficult and computationaly expensive to implement.

## Translating SyCoL

Translation differs from interpretation by the fact that it does not actually execute the program, but rather it translates the program into another language so that the program may be subsequently interpreted or compiled. Therefore, real-time constraints only enter the picture when the consideration of the language to translate to is made. Ideally, the language to translate to would be an efficient one like C or Pascal and not an interpreted language. However, an interpreted language could be chosen if executes under the minimum real-time constraints.

The translation process is basically a mapping process. It starts, as with interpretation, with reading the source file. As it is reading the source file into memory, it takes the statements from the source file and looks up the equivalent statements in the target language. After finding the equivalent statements, some translators perform some optimazation new statements, removing inefficiencies that may have arisen from the translation process. After this, the target file containing the translated statements is written out. This target file may then be executed by an interpreter or fed into a compiler to yield an executable program (Aho 114).

## Compiling SyCoL

The final method of implementing SyCoL is compilation. The compilation process is much the same as the translation process, except that the target language is the machine language of a target computer. This method yields the slowest development time, but at the same time, yields the quickest execution time.

The process is the same as the translation process with some additional points added. First, after the target file is written out the compiler then reads it back in and converts it into machine code readable by the computer. This machine code is then combined with existing libraries of machine code to form an executable program. The entire process takes a great deal longer than simple translation, but yields a program that may be executed extremely quickly. The programmer then executes the program, notes the errors, and goes back to the source file and makes changes to fix the errors. After the errors are corrected, the source file must be re-compiled, and thus program development time is extended (Aho 22).

## Comparison of Implementations

Given the above information, the implementation choice must be made. To interpret SyCoL would yield a system in which program development would be lessened greatly. However, real-time system considerations must be considered, and thus, interpretation is out. Compiling the language would result in very efficient execution time, however, program development time would be extended greatly. Before accepting or eliminating compilation, another important consideration must be examined: portability. Compiler writing is inherently a machine dependent process, that is, one written, the compiler will only run on one type of computer system. Thus I believe that translation would be the best choice in light of the needed compromise between programming ease and economic considerations of implementation. I think that the best choice of a target language would be C, mainly because of its ability to express low-level activities easily and efficiently.

# Chapter Six

## Conclusions and Thoughts About SyCoL

This chapter concludes the paper by giving a summary the research, and continues by outlining some of my thoughts about the affect of SyCoL on the future of industry.

## Summary of SyCoL Research

This research has attempted to unite the fields of software engineering and control system design. I have extracted what I believe to be the essential concepts for the design of a new distributed PLC programming language from existing language designs. I have also taken the principles of real-time software engineering and applied them to PLC programming. By combining these two fields, I believe I have created a viable language, a language to bring the control aspect of manufacturing in line with state of the art of other manufacturing technologies.

The research began by examining a number of books on PLC programming, and outlining the differences between the PLC programming languages. I then examined some of the critiques of the languages and then evaluated them myself. At the end of this process, I had gathered an extensive list of what a PLC language should and shouldn't have. After this I looked into a number of books on software engineering, and into my own class notes on the subject. From these sources I compiled another list of the needs that the designer of a real-time system needs to address. With these two lists in mind, I examined the basic idea that I had for a new language and modified it to conform to the needs that I had extracted.

Thus I believe that my system is a fair compromise

between existing PLC languages and recommended software engineering techniques. I think that the system would stand up both to the scrutiny of the industrial engineer and the computer scientist.

## Thoughts About SyCoL

I believe that once a SyCoL system is implemented it will prove itself as a viable language very quickly. I think that the design of the language allows both professionals and students to use it to the fullest. I believe that the language will make its biggest impact in the medium-scale job shops due to the quick and efficient program development and execution of the language. I think that the owners of small-scale jobs shops would find that it might be more efficient to use other programming methods.

## Works Cited

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1988. Compilers, Principles, Techniques, and Tools. Reading: Addison-Wesley.

Barney, George C. 1988. Intelligent Instrumentation. New York: Prentice Hall.

Johnson, David G. 1987. Programmable Controllers for Factory Automation. New York: Marcel Dekker.

Lewis, T.G. 1991. CASE: Computer-Aided Software Engineering. New York: Van Nostrand Reinhold.

Lukas, Michael P. 1986. Distributed Control Systems. New York: Van Nostrand Reinhold.

Sodhi, Jag. 1991. Software Engineering. Blue Ridge Summit: TAB Professional and Reference Books.